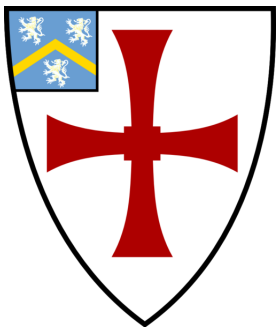
M.Sc. Thesis

Artificial Intelligence Techniques Applied To Draughts

Supervisor: Dr. Magnus Bordewich

17/02/2013



Department of Engineering and Computer Sciences

University Of Durham

ABSTRACT

This thesis documents the work done to develop a draughts playing program that learns game strategies utilising various Artificial Intelligence (AI) techniques with the goal of being able to play draughts at a reasonably high skill level as a result of having played against itself without external guidance.

Context/Background:
AI is a fast evolving field of study. The motivation being programming computers to learn from experience should eventually eliminate the need for this detailed, time consuming, and costly programming effort currently required to program solutions to problems.

Aims:
The aim is to investigate a variety of AI techniques. The program's effectiveness will be assessed in both evaluating moves and playing a computationally intensive game. Minimax based algorithms together with a basic scoring heuristic are used to evaluate enough of the game tree to pick high utility moves. Later the scoring heuristic is augmented using artificial intelligence techniques. As a result of this training "smart scoring behaviour" the program is expected to learn how to best assign values to each of the squares on the draughts board enabling it to play at an adequately high skill level.

Method:
In this thesis a version of the board game Draughts is implemented in the Java programming language. Players were developed using a variety of techniques. These algorithms were tested by comparing running times, number of nodes of the game tree searched and the utility of the moves picked. In addition an algorithm is developed to assign scores to given board states using a genetic algorithm.

Results:
The project was a success for the most part permitting the creation of the game of draughts in the JAVA programming language. Four out of the five proposed move selection techniques were successfully tested in isolation. Finally the genetic algorithm demonstrated the ability to augment the scoring heuristic without the benefit of external guidance in the form of human experience.

Keywords – Draughts, Checkers, Artificial Intelligence, Two Player Games, MiniMax, Alpha beta Pruning, Ant Colony Optimization, Genetic Algorithms

ACKNOWLEDGMENTS

I would like to thank the Engineering and Computer Science Department at Durham University for providing me the means and opportunity to complete this Master of Science in Computer Science degree.

In particular, I would like to thank Dr Magnus Bordewich from the University Of Durham for his time and support supervising me for the duration of this research project. His guidance and knowledge in the field was paramount in the writing of this thesis.

I would also like to thank my family and friends for their endless support and encouragement throughout this degree.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

1.1 Background to Computer Game Playing

Artificial Intelligence has been a fast evolving field of study since its conception as the focus of a 2 month conference at Dartmouth College in the summer of 1956. The conference was attended by an array of people from both academia and industry including John McCarty, Trenchard More, Arthur Samuel, Ray Solomonoff, Oliver Selfridge, Allen Newell and Herbert Simon this group went on to lead the field [21]. The term 'Artificial Intelligence' was created by John McCarthy to describe the field of study focused on during the conference at Dartmouth College as 'the science and engineering of making intelligent machines, especially intelligent computer programs' [17]. The motivation being programming computers to learn from experience with the long term aim of eliminating the need for detailed, time consuming, and costly programming that is currently required to program solutions to problems [22]. This in turn would allow a shift in the equilibrium of a bistable system described thusly by Newell: 'we could design more intelligent machines if we could communicate with them better; we could communicate with them better if they were more intelligent' [18].

Artificial intelligence techniques developed since 1956 have been practically applied in machine translation e.g. Babelfish, speech and handwriting recognition, and on several Mars rovers e.g. Spirit and Opportunity. The techniques have also been utilised with an emphasis on creating computer programs that play games. Claude Shannon was one of the first to apply Artificial Intelligence techniques in this context. He wrote the first pa-

per detailing a chess playing program titled 'Programming a computer for playing chess' at a time when computers were not widely available, and were woefully underpowered compared with today's machines [27]. Since then many others have felt inspired to combine Artificial Intelligence methods, such as the game-tree search algorithm described by Shannon, with knowledge of game strategies to create programs that play other computationally complex games such as Go, Chess, Draughts and Othello. There has been varying levels of success, owing to the constantly evolving techniques, and constantly improving hardware available. Only relatively recent advances in Artificial Intelligence and hardware speed have allowed computers to win games against grandmasters, such as those games played between Garry Kasparov and Deep Blue in 1996.

1.2 Objectives

In this thesis an intelligent draughts playing program is created using moderately powerful hardware to compare Artificial intelligence techniques using the board game draughts. This is done by first testing six move-selection techniques in isolation, comparing running time, number of nodes in the game tree searched and move utility. The first of these move selection techniques picks moves at random and will be used as a control. Other move selection techniques include the MiniMax algorithm and other approaches derived from the original MiniMax algorithm which seeks to improve its performance by utilising Alpha Beta Pruning and move ordering. The fifth is an Ant Colony Optimisation algorithm adapted to play draughts. Additionally a genetic algorithm is implemented to learn a heuristic evaluation function for learning game states.

The following requirements have been identified as needing to be completed if the objective of creating an intelligent draughts player is to be fulfilled. The first and arguably the most important requirement is to implement the game of draughts utilising the JAVA programming language. The game must work correctly as per the accepted rules of draughts, including any local variants of the rules. In addition a testing framework

should be implemented to facilitate persistence so that results obtained from the program can be saved and accessed after the program ceases to function. The data saved by the testing framework should permit the extraction of meaningful data from the draughts game that can be used to identify any problems with the working of the program, and facilitate the comparison of AI techniques.

The second objective is to implement a variety of AI players. These AI players will offer a preferable alternative to a naïve brute force algorithms that attempts to search the entire game tree to find a winning strategy. Exhaustive search is basically infeasible; as such a variety of AI players has been implemented which pick moves quicker, using minimal computational effort while picking moves with high utility.

The final objective is to augment the scoring heuristic using artificial intelligence techniques without the benefit of external guidance in the form of human experience. This will be achieved by having the program play against itself. As a result of this training it is expected that the program score game states more accurately and assign a value to each of the squares on the draughtboard enabling the program to play the game at a higher skill level. This will be achieved by a genetic algorithm.

1.3 Thesis Structure

The rest of this thesis is structured as follows:

Section 2 explains the reasoning behind choosing the game of Draughts as being sufficiently interesting to be used as a test bed for the chosen artificial intelligence algorithms. It also details the general rules of Draughts along with the specific variants of the rules that will be implemented. Additionally details are provided of the evaluation function used to assign scores to the draughtboard so that the reader of this thesis is familiar with it going into the subsequent chapters.

In sections 3 and 4 an introduction is given to the three differing AI techniques namely MiniMax, Alpha-Beta Pruning and Ant Colony Optimization. In addition details are

given of how they have been implemented in the project, details of the experiments carried out using each of them, the expected outcomes of these experiments and finally the results obtained from the aforementioned experiments.

In section 5 genetic algorithms are introduced. Details are then given of the genetic algorithms implemented to learn an improved evaluation function and the experiments carried out to test it, our predicted results for each of the proposed experiments, and the results.

In section 6 implementation specific details of the draughts program implemented in this project are given. The details include the class structure, systems architecture, tools used and a more detailed description of each of the algorithms implemented in the draughts program.

In section 7 - Evaluation - This section provides some analysis of the results found in each chapter of the thesis.

In section 8 - Conclusion - This section provides a summary of the thesis's research suggesting avenues for further work within the field.

2 Draughts Game

2.1 Draughts

2.1.1 Why study draughts?

In terms of classical game theoretic terminology, the two player game draughts can be defined as a Finite Perfect-information Extensive form game. Finite means the game is always finished in finitely many moves. Perfect-information means that at each move every player knows the current state of all of the actions and what they can do. Extensive form means the game can be represented as a tree as follows:

More formally the game is a tuple $G = (N, A, H, Z, \chi, \rho, \sigma, u)$ where [14]:

- N is a set of n players;
- A is a (single) set of actions;
- H is a set of nonterminal choice nodes;
- Z is a set of terminal nodes; disjoint from H;
- $\chi : H \to 2^A$ is the action function, which assigns to each choice node a set of possible actions;
- $\rho : H \to N$ is the player function, which assigns to each nonterminal node a player $i \in N$ who chooses an action at that node;

- $\sigma : H \times A \to H \cup Z$ is the successor function, which maps a choice node and an action to a new choice node or terminal node such that for all h_1, $h_2 \in H$ and $a1$, $a2 \in A$, if $\sigma(h_1, a_1) = \sigma(h_2, a_2)$ then $h_1 = h_2$ and $a_1 = a_2$; and
- $u = (u_1, ..., u_n)$, $where\, u_i : Z \to \mathbb{R}$ is a real-valued utility function for player i on the terminal nodes Z.

While the condition $\sigma(h_1, a_1) = \sigma(h_2, a_2)$ allows for the game to be defined as extensive form it means that game states that are from different parents but are considered identical in game play are considered to be different. This results in a game tree larger than that of the game space having to be evaluated. However, being an extensive form game allows the procedure of playing the game to be described as a game tree which is a directed tree graph used in the analysis of strategies for a game. The vertices represent game states, the edges represent moves that allow a player to move between states and the leaves represent terminal game states over which each player has a utility function. In a complete game tree all the possible moves from each game state are included at each vertex starting from the root of the tree. Contrary to the n players proposed above in the general game definition, the game of draughts is played between two people, as such two levels of this tree represent one move, consisting of two half-moves i.e. a turn by black and a turn by white, each of which is called a ply [21]. The underlying tree is as follows:

- Nodes $= H \cup Z$
- Edges $= u \to v \leftrightarrow \exists a \in A\, s.t.\, \sigma(u, a) = v$

The game of draughts is a constant-sum game. This means that there exists a constant C such that for all $z \in Z$, $\sum_{i \in N} u_i(z) = C$. In other words the total utility is constant, and one player's gain (in utility) is always another player's loss. Furthermore, draughts can be referred to as being a zero-sum game because the constant C is equal to 0, because there are only three types of terminal states: those in which black wins, white wins

and there is a draw, which give utility (+1,-1), (-1,+1) and (0,0) for black and white respectively [14].

The reasons for choosing the game of draughts to study include that it is a game with simple well defined rules which allows for greater emphasis to be placed on the learning techniques. However, despite having simple well defined rules computing a winning strategy is difficult. This is demonstrated by the fact that the game of draughts has up until this point only been weakly solved. Being weakly solved means that it has been shown how each player can play optimally, and that if each player does so without making any mistakes then the game will end in a draw. However, if either player deviates from this cycle of playing perfect moves then nobody can be certain which is the best move to choose. This is due to fact that computing the outcome for the complete game tree is too computationally intensive due to its size. The game tree complexity, defined as the number of nodes in the game tree, is 10^{31} positions [4]. At 1000000 moves per second would take $3*10^{17}$ years to evaluate. By comparison the universe is only about 4×10^{17}years old. When played on an n×n grid the game of draughts is EXPTIME-COMPLETE. EXPTIME denotes the class of languages that is solvable by a deterministic Turing machine in $O(2^{p(n)})$ time, where p(n) is a polynomial function of n, in other words EXPTIME = $\bigcup_{k\in\mathbb{N}}$ DTIME (2^{n^k}) [29]. EXPTIME-COMPLETE denotes a decision problem in the class EXPTIME to which every decision problem in EXPTIME is polynomial time reducible. As such they are considered to be the hardest problems in EXPTIME [29]. The alternative to computing the complete game tree is to compute the outcome for part of the game tree to a fixed depth and then evaluate which game states are preferable using a heuristic function. This is the approach of the MiniMax algorithm. However, this approach results in imperfect information about the consequence of having picked a move [24].

This is in contrast to simpler games such as tic-tac-toe which is trivially solvable with a state space complexity of 765 different position and a game-tree complexity of 26830

nodes[25], and can be shown to be PSPACE COMPLETE[20]. PSPACE denotes the class of languages that are decidable in polynomial space using a deterministic Turing machine, in other words PSPACE = $\bigcup_k$SPACE (n^k) [29]. PSPACE COMPLETE denotes a decision problem that is both in PSPACE and to which every decision problem in PSPACE is polynomial time reducible. As such they are considered to be the hardest problems in PSAPCE [29]. Similarly connect four has been solved and is in PSPACE and has a game-tree complexity of at most 10^{13} nodes [2]. One factor which makes these games simple and implies they are in PSPACE is the maximum number of moves of the game is polynomial in the size of the board.

If the maximum number of moves is polynomial in the size of the board then you are in PSPACE whereas if it is exponential then you are in EXPTIME. Thus the game of draughts has more in common with games such as Chess and Go which are partially solved and EXPTIME-COMPLETE. For Chess the estimated lower bound on the game-tree complexity was calculated thusly $p(40) \approx \frac{64!}{32!(8!)^2(2!)^6} \approx 10^{43}$ by Claude Shannon, a number known as the Shannon number [27]. Go has an average of 150 moves a game, with an average of about 250 choices per move, so has an estimated game-tree complexity of 10^{130} [2]. Thus all three games are too computationally intensive to solve by brute force due to the limitations of present day hardware.

2.1.2 Variants of Draughts

Draughts is played on a draughtboard which is divided up into an equal number of squares width ways and length ways. The size of the draughtboard used varies it can be either 8×8, 10×10 or 12×12. These squares alternate in colour between a light colour and a dark colour, with squares of the same colour being diagonally adjacent to each other. A square on a draughtboard can be occupied in 5 different ways: either it is empty or it is occupied by one of two kinds of piece, namely a 'man' or a 'king' each of which can be one of two different colours, namely 'white' and 'coloured'. At the start of a game of

draughts each of the two players is assigned one of the different coloured sets of pieces to legally move. The players take it in turns to move one of their pieces. These pieces can be moved legally in one of two ways:

1. A piece can perform a non-attacking move. This is done by moving the piece into an empty diagonally adjacent square. The opponent then begins their turn.

2. A piece can perform an attacking move. This is done by moving the piece such that it jumps over an opponent's piece that is occupying a diagonally adjacent square. Thus the piece moves a distance of 2 squares landing in the diagonally adjacent square on the opposite side of the opponent's piece. After the jump has been completed the opponent's piece is removed from the board. The opponent can continue to take their opponents pieces in this fashion in successive moves, else the opponent begins their turn

In both of these cases men must always move towards their opponent's end of the board. Thus they are limited to moving into one of the two diagonally adjacent squares that see them advancing towards their opponents end of the board, whereas kings are free to choose to move into any of the four diagonally adjacent squares. If a player's piece successfully reaches their opponents side of the board then that piece is 'crowned' i.e. marked in such a way as to distinguish it from one of their 'men', and it becomes a 'king'. The number of pieces that each player starts with depends on the size of the draughtboard being used and can be 8, 12 or 16 on a board of size 8×8, 20 on board of size 10×10 or 30 on a board of size 12×12.

The game of draughts has numerous variants as played in several countries these include changes to which player starts first i.e. in some variants white starts first and in others the coloured pieces start first. In draughts making the first move can be an advantage because it can allow you to be first in creating a weakness in your opponent's position. However, the advantage of going first in draughts is much less than that offered when

going first in chess. This is due to the fact that by going first you can be forced to first open weakness in your own position either by lack of skill or by running out of safe moves, or you can just run out of legal moves first and have nowhere to move your pieces [16]. What follows is a description of each of the variations of the rules of draughts with analysis of the effect on the computational effort needed to play the game of draughts.

Changes can be made to how pieces can move i.e. in some variants 'kings' are not limited to either moving to diagonally adjacent squares or jumping over enemy pieces but can instead move any number of squares in a diagonal direction to take a piece, limited only by the edges of the game board, similarly to a bishop in a game of chess. This variant of the rules results in the game taking a fewer number of moves to be completed as the pieces are limited to moving a certain number of squares per turn. This reduced number of moves sees a reduction in the number of levels in the game tree, and thus a reduction in the overall size of the full games tree when compared to games which do not use this variant of the rules. Thus games using this variant of the rules are slightly less computationally intensive.

Changes can result in a player having no choice but to attack when the opportunity arises e.g. some variants of draughts dictate that attacking moves must be chosen in priority to non-attacking moves when they occur. This more aggressive style of play sees the number of pieces on the board reduced at an accelerated rate. The accelerated reduction in the number of pieces on the board sees an accelerated reduction in the branching factor as fewer pieces in the board equates to less choices of pieces for the player to move. Therefore the full game tree for games that enforced this variant of the rules has a smaller number of game states when compared to games which do not use this variant of the rules. As a result of the average branching factor being reduced, it follows that regardless of the board size, games using this variant of the rules are typically going to be less computationally intensive.

Changes can be made to how pieces may attack e.g. some variants of draughts dictate that attacking moves must result in the player taking all of their opponents pieces that they can given the attacking move that they have just chosen. This more aggressive style of play sees the number of pieces on the board reduced at an accelerated rate, resulting in a reduction in the branching factor of the game tree as fewer pieces on the board equates to less choices of pieces for the player to move. In addition the consequences of an attacking move make it mandatory to make a second, third or fourth etc. also equates to less choices of move for the player to pick from. Therefore the full game tree for games that enforced this variant of the rules has a smaller number of game states when compared to games which do not use this variant of the rules and as such as are slightly less computationally intensive.

2.2 English Draughts

Throughout the development of the draughts game the aim was to emulate the game of draughts as it is played according to English variant of the rules. This dictates that the game be played on an 8×8 grid, with each of the two players starting with 12 men, with the coloured player having the first move. The game is set up as shown in Fig 2.1 [32].

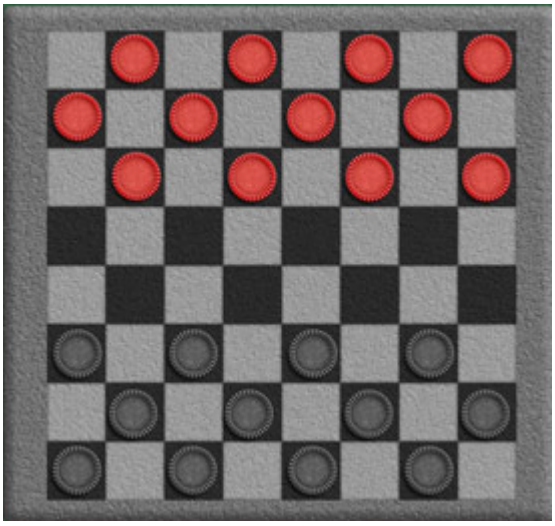


Figure 2.1: English Draughts Starting Arrangement

In the English variant of draughts the game ends when one of three conditions are met. A game ends when either side has run out of pieces, is unable to make any more legal moves or has moved 40 times without capturing an enemy piece or crowning one of their own pieces. Additionally the English variant of the rules specifies that attacking moves take precedence over non-attacking moves, mandating that where attacking moves arise they should be taken. If more than one attacking move should arise the player will have a choice as to which move to execute. The English rules further state that once an attacking move has been started i.e. the first jump has been made then the player is forced to finish the move taking as many of the other opponents piece as they are legally allowed to. The only exception to this rule is if one of a player's men has been crowned after which their turn automatically finishes. These additional rules are favourable because they all have the effect of reducing the size of the game tree in different ways.

Forcing attacking moves causes the player to have fewer choices of moves when attacking moves are available. For example at the beginning of a game of draughts (where there are no kings on the board and there is are attacking moves) the player has 7 or

8 moves to pick from, however when there is an attacking move (and thus the rules for the player to pick from one of them to proceed with the game) the player only has 1 - 2 moves to pick from. As such, taking a game tree of depth 5 as an example, this can see the size of the game tree that needs to be evaluated reduced from $8^5 = 32732786$ nodes to $2^5 = 25$ nodes. This makes the game tree quicker to search through when trying to ascertain the best move, although just occurring at a few places in the game tree it is not a significant enough reduction in the game tree to allow the full game tree to be searched without extensive computational effort.

Forcing a player to take as many of their opponent's pieces as possible reduces the number of turns required to complete a game. This occurs because forcing the player to take their opponents pieces at the earliest opportunity accelerates the rate at which pieces are removed from the board when compared with if the player is given the choice to delay taking their opponents pieces. The effect of this is to accelerate the rate at which both players run out of pieces which sees the game end in a reduced number of turns. This was demonstrated when we ran 100 games using this forced attacking rule which ended in an average of 74 moves each, and 100 games without forced attacking as a rule which ended in an average of 182 moves each. Thus we found that that games using the forced attacking rule ended on average 108 moves quicker than games not using this rule. Despite the program never attempting to traverse the full game tree, and the reduction of the game tree as a whole having negligible effect on the portion of the game tree that is searched in order to find the best move, it is interesting to note that this reduction in the size of the game tree as a whole can facilitate the simulation of games at a faster rate.

Having the game terminate when either player goes 40 moves without capturing or crowning a piece also reduces the number of turns it takes for a game to finish. This reduces the size of the game tree from having a possible infinite number of levels to a finite number as players are penalized for just endlessly moving their pieces around the

board.

The goal is to have the game of draughts played by a variety of players each utilising different algorithms commonly found in the field of Artificial Intelligence. These algorithms will include a player who picks random moves (which will be used as an experimental control) a MiniMax algorithm (which will traverse the game tree to a given depth), an Ant Colony Optimisation algorithm (which simulates foraging behaviour in the game tree to look for good states), and a genetic algorithm to learn the evaluation function, the details of which are expanded upon in the coming chapters.

2.2.1 Scoring Heuristic

Previously it was identified that the game of draughts is a deterministic, turn taking, two-player, zero-sum game of perfect information. However, despite the game of draughts being deterministic, and thus the entire search space is known to us, time does not permit an exhaustive search as such there exists no known algorithm which will predict the best move short of the complete exploration of every acceptable path that comprises the complete game tree. Lacking time for such a search, heuristic procedures must be depended upon [23]. Details will now be given of the linear polynomial evaluation function used to assign scores to the draughtboard so that the reader of this thesis is familiar with it going into the subsequent chapters.

A square on a draughtboard can be occupied in 5 different ways: either it is empty or it is occupied by one of two kinds of piece, namely a 'man' or a 'king', each of which can be one of two different colours, namely 'white' and 'coloured'. Thus, the state of each of the 64 squares can be specified by giving an integer from 0 to 4. The positions of the pieces in the game of draughts are stored using an 8×8 matrix and a numerical representation of what piece is on which square: empty squares have value 0, white men have value 1, coloured men have value 2, white kings have value 3 and coloured kings have value 4. In the interest of efficiency only 32 of those 64 values are stored as only half

of the board squares can ever be occupied by a piece because they initially occupy board squares of the same colour and throughout the course of the game are only ever moved into diagonally adjacent positions. In addition to the above 8×8 matrix, an entirely separate 8×8 matrix is used to keep a numerical representation of how desirable it is to occupy each of the squares that make up the board. Again this matrix is used to store a total of 32 values as only half of the board squares can ever be occupied by a piece and thus it only makes sense to assign a value representing the desirability to the squares that will be occupied during the course of the game.

The basic scoring function is calculated by taking the sum over all white pieces multiplied by the value of the square that they are currently occupying and then subtracting the same value but for coloured pieces. A small random weighting is then added to the score this is to prevent the move picking algorithms (except for the player who picks random moves) from picking the same move repeatedly when given the choice between several moves of equal utility. The small random weighting is small enough such that it doesn't influence the picking of a move except in situations where the moves would have had exactly the same utility. As a result of preventing the same move being picked repeatedly it also prevents the same game being played repeatedly. It is desirable to avoid this, since if the opponent wins one game he could play the same variation and win continuously [27].

The program makes use of the following basic scoring function, which is referred to as the heuristic evaluation function:

$$White\,Score = \sum_{ij}(W_{ij} + WK_{ij})B_{ij} - (C_{ij} + CK_{ij})B_{(8-i)(8-j)} + r$$
$$Coloured\,Score = -White\,Score$$

Where W_{ij}, WK_{ij}, C_{ij}, CK_{ij} are derived from the piece positions and are defined as follows:

W_{ij}= 1 iff square has a white man

WK_{ij}= 2 iff square has a white king

C_{ij}= 1 iff square has a coloured man

CK_{ij}= 2 iff square has a coloured king

In addition B is a matrix giving weights to different squares. While conducting the experiments in the chapters MiniMax and Ant Colony Optimisation the matrix B was initialised to have each of its values set to 1 and was never changed during the course of those experiments. Only during the experiments detailed in the chapter about genetic algorithms was the scoring matrix Matrix B modified by the genetic algorithm. Finally r is the small random weighting discussed above.

3 MiniMax featuring Alpha-beta Pruning

3.1 MiniMax

In this chapter the first approach to selecting a move given a current game state is examined. It is not satisfactory to look at the children of the current game state in the game tree and pick the move corresponding to the board position with the highest heuristic score. The heuristic is to simple to accurately predict longer term effects and will certainly never sacrifice a piece in order to build a stronger position. Looking an arbitrary number of moves ahead, finding the best board position and then picking a move leading towards that state will do no good either because reaching that position would require the cooperation of the opponent. The game of draughts is a non-cooperative game in which our adversarial opponent is actively trying to work against us [22]. The adversarial opponent will choose alternative moves leading away from the state good for the other player. When assessing a move our opponents response should also be considered, and what the worst case scenario is i.e. the responding move by the opponent that minimises our future chances. This idea is captured in the "Minimax" algorithm. Therefore the MiniMax algorithm described below is used to planning ahead in a world where other agents are planning against us [21].

The MiniMax Algorithm was first proposed by Neumann & Morgenstern in 1928 for two-player zero-sum games and is the same as the Nash equilibrium [31]. It is a recursive algorithm used to find the best choice from all the available legal moves that can be played from the current board state in an n-player game [5]. The recursion proceeds all

the way down the game tree via all the legal moves and counter-move to the leaves of the tree, and then the MiniMax values are backed up through the tree as the recursion unwinds, evaluating the best move on its turn and the worst move on its opponent's turn this is shown in algorithm 3.1 which was adapted from [21]. The algorithm earned its name 'MiniMax' from this constant alternating between picking the move with maximum utility on its turn, and the move with minimum utility on its opponent's turn.

Algorithm 3.1 MiniMaxPlayer pseudocode algorithm

```
minimax(game state, depth)
    if depth <= 0 then
        return heuristicEvaluationFunction(gameState)
    end

    local alpha = -gameState.player×INFINITY

    local gameState = next_gameState(game state, nil)
    while gameState ~= nil do
        local score = minimax(gameState, depth-1)
        alpha = gameState.player == 1 and math.max(alpha, score) or math.min(alpha,
score)
        gameState = next_gameState(gameState, child)
    end
    return alpha
end
```

The MiniMax algorithm could permit a perfect game to be played if it was allowed to consider the complete game tree and establish which of the leaves ends in a win, lose or a draw. By working backward from the end it can be determined at any position whether there is a forced win, the position is a draw or is lost. However, even with high computing speeds it can be shown that this computation is impractical because the number of game states that require examination is exponential in the depth of the tree [21]. This can be further highlighted by the following calculation. In the beginning of a typical game of draughts (where the lack of kings reduces the number of possible moves by up to half)

there will be of the order of about 8 legal moves per turn. Games of draughts last on average for 70 turns. This implies there are at least $8^{70}(= 10^{63})$ moves needing to be evaluated from the initial position [27].

However, a heuristic evaluation function is utilised to give values to non-final game states in order to stop tree construction when a certain depth has been reached, allowing us to search a much smaller truncated game tree instead of the complete game tree. This is preferable to the naive approach since it requires analyse of much fewer game states when picking the best move. Owing to the fact the game of draughts in a zero sum game, if the evaluated utility value is negative this indicates an outcome that is better for the min player. If the value is positive this indicates an outcome that is better for the max player. Values close to zero indicate that the outcome gives little benefit to either player. An example is given to demonstrate how the MiniMax algorithm is used when evaluating the example game tree depicted in Fig3.1.

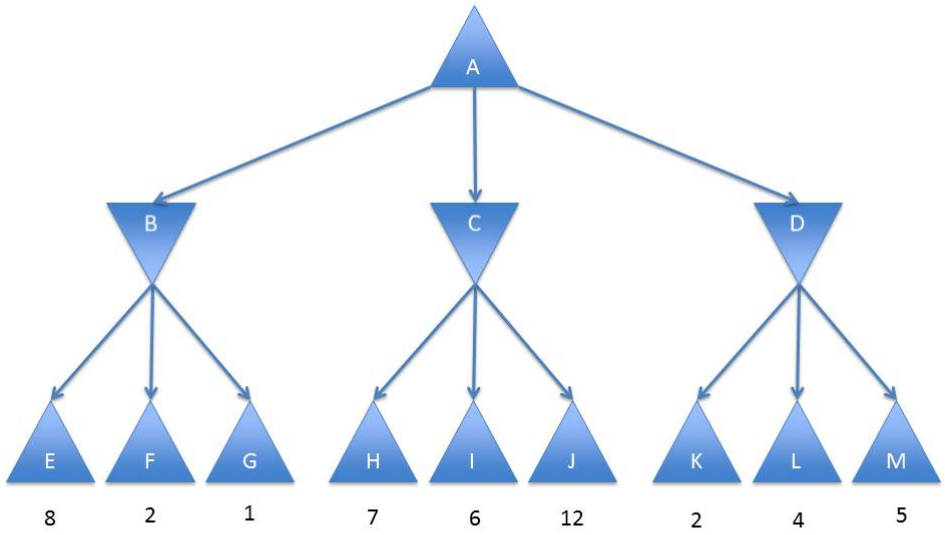


Figure 3.1: Game tree with leaf node utilities

The game's current state is the root of the tree. Each of the player's valid moves from this initial state become a child of that root, then each of the opponent's moves in response to the player's moves become children of those children, and so on constructing a tree of board states. In the example the game tree is curtailed at a depth of 3 at which point the leaves are given utilities. A set of utilities has been fabricated for leaf nodes in the game tree for the purposes of this example, in our implemented draughts program the utility of each position will be calculated by the heuristic evaluation function described in the previous chapter. The triangles pointed up represent maximising nodes: these are nodes that given a choice between utilities always pick the one with the highest value. The triangles pointing down represent minimising nodes: these are nodes which given a choice between utilities always pick the one with the lowest value. The Minimax Algorithm is applied using the following steps:

Step 1: Start at node A (the root) and initiate a depth first search of all of the nodes.

Step 2: Arrive at minimising node B from maximising node A.

Step 3: Continue from minimising node B to maximising node E.

Step 4: Node E has a value of 8, propagate this back up to minimising node B and since node B has only had 1 child evaluated at this point its value becomes 8 by default.

Step 5: Visit node F which has a value of 2. Propagate this back up to minimising node B and compare it with the current value of node B which is 8, because node B is a minimising node it prefers 2 over 8 and so its value now becomes 2.

Step 6: Visit node G which has a value of 1. Propagate this back up to minimising node B and compare it with the current value of node B which is 2. 1 is less than 2 and so because node B is a minimising node it prefers 1 over 2 and so its new value becomes 1.

Step 7: Node G was the last child of node B and so the value of minimising node B is propagated backwards up the tree to node A whom has only had 1 child evaluated

at this point and so its value is set to 1 by default. The Minimax algorithm continues performing the steps in this fashion exploring the other sub trees.

The Minimax algorithm continues performing the steps in this fashion exploring the other sub trees.

Step 8: The second subtree is then evaluated starting with Node H is explored and its value of 7 is propagated this back up to node C whose value becomes 7 because it has only had 1 child evaluated at this point. Node I is then explored and its value of 6 is propagated back up to minimising node C and becomes its value because 6 is smaller than 7. Node J is then visited it has a value of 12. This value is propagated back up to minimising node C which keeps its value of 6 as this is smaller than 12. The value of node C is then propogated backwards up the tree to maximising node A and becomes its new value as 6 is bigger than 1.

Step 9: The third subtree is then evaluated starting with Node K is explored and its value of 2 is propagated back up to node D whose value becomes 2 as it has only had 1 child evaluated at this point. Node L is then explored and its value of 4 is propagated back up to minimising node D which keeps its value of 2 because 2 is smaller than 4. Node M is then explored and its value of 5 is propagated back up to minimising node D which keeps its value because 2 is smaller than 5. Node M was the last child of node D and so node D's value is propagated backwards up the tree to maximising Node A whose values stays at 6 because 6 is bigger than 2

Step 10: Thus the best move for the player to make is to pick move C.

As shown in the above example the MiniMax algorithm performs a complete depth-first exploration of the game tree. The effective branching factor of the tree is the average number of children of each node which is equal to the average number of legal moves in a position. The number of nodes to be explored usually increases exponentially with the number of plys. Therefore if the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$.

The space complexity is $O(b^m)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time. It is therefore impractical to completely analyse games such as draughts using the MiniMax algorithm [21].

Two move picking algorithms have been created, the first is the RandomPlayer a control. This player selects a random move from the selection of legal moves available from the current game state. The algorithm does this by compiling a list of all possible moves the player can make from the current game state. The algorithm then chooses a move uniformly at random from this list. This algorithm has no changeable variables.

The second is the MiniMaxPlayer. This player uses the MiniMax algorithm in conjunction with a heuristic scoring function (defined in the previous chapter) to select a move that maximises the minimum gain as defined by the scoring heuristic. The move picked will be from the selection of legal moves available from the current game state. The algorithm does this by being passed the current board state and the depth of termination. It then searches the game tree to the desired depth, evaluates terminal states using the heuristic and picks the best move using the MiniMax algorithm. The only variable is "depth", which determines the maximum layer in the game tree searched to. Increasing this variable allows the player to search more of the game tree. This increase in the number of game states visited increases the player's information about the game tree which allows them to make better moves. Increasing the depth searched is done at the cost of amplified computational expense and longer running times. This is owing to the fact that the number of nodes that need to be searched is exponential in the depth searched. Therefore even a small increase the depth by as little as one has a profound impact on the total number of nodes that need to be searched. This huge increase in the number of game states that need to be searched sees a huge increase in the amount of computation effort needed to do this and also requires a huge amount of time to complete.

3.2 Alpha-beta Pruning

Alpha-Beta Pruning is a method of speeding up a MiniMax Algorithm. Pruning to allow deeper search was introduced by John McCarthy during the Dartmouth Conference in 1956. The algorithm is an augmentation to the MiniMax algorithm that prevents the exploring of the branches of a game tree which are not of further interest to the player or to his opponent [23]. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. The trick is that it is possible to compute the correct MiniMax decision without looking at every node in the game tree unlike other heuristic pruning methods which aren't always guaranteed to give the same result as the unpruned search. It is this ability to prune away the branches of a search that cannot possibly influence the outcome of the final decision that allows it to have a much smaller running time than that of the MiniMax algorithm. [21]

The algorithm derives its name from the fact it maintains two values, alpha and beta that describe bounds on the best values that appear anywhere along the path from the root to the currently evaluated node. If a max node already has a value of 10 and the next child (min node) to be evaluated another child has a value at most 9, then this child need not be evaluated any further as it will not be chosen by the max node anyway. The parameters α and β are defined as:

α = the value of the best (i.e., highest-value) choice found so far at any choice point along the path for MAX and is initialised to negative infinity.

β = the value of the best (i.e., lowest-value) choice found so far at any point along the path for MIN and is initialised to positive infinity.

The parameter α represents the maximum score that the maximising player is assured of which must be exceeded for a board to be considered desirable by the side about

to play. The parameter β represents the minimum score that the minimising player is assured of which must not be exceeded for the move leading to the board to have been made by the opponent [23]. The Alpha Beta algorithm 3.2 updates the values of α and β as it searches the game tree using depth first search and as it does so the gap between the value of these two variables decreases. When beta becomes less than alpha, it means the current position cannot be the result of best play by both players and need not be explored further. At this point that the algorithm ignores or "prunes" these branches of the search tree that would yield less favourable results, thus saving time [21]. Alpha-beta pruning assumes that the analysis of all branches is otherwise carried to a fixed ply depth and that all board evaluations are made at this fixed ply level [23].

Algorithm 3.2 AlphaBetaPlayer Pseudocode Algorithm

```
alphabeta(game state, depth, α, β, Player)
    if depth = 0 or node is a terminal game state
        return the heuristic value of game state
    if Player = MaxPlayer
        for each child of game state
            α := max(α, alphabeta(child, depth-1, α, β, not(Player) ))
            if β ≤ α
                break (* Beta cut-off *)
        return α
    else
        for each child of game state
            β := min(β, alphabeta(child, depth-1, α, β, not(Player) ))
            if β ≤ α
                break (* Alpha cut-off *)
        return β
```

As an example it is next demonstrated how the MiniMax algorithm with alpha-beta pruning is used when evaluating the example game tree depicted in Fig3.2:

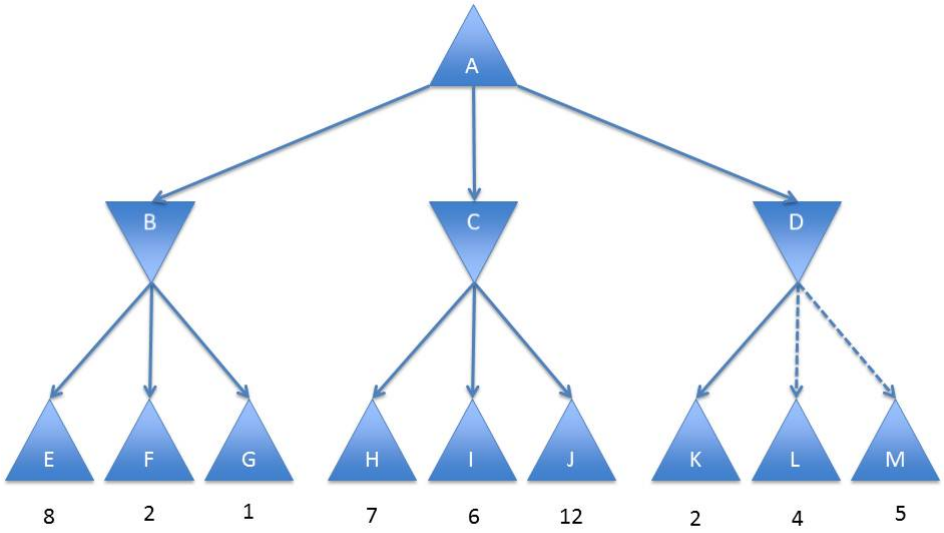


Figure 3.2: Game tree with alpha-beta pruning with leaf node utilities

The triangles pointed up represent maximising nodes. The triangles pointing down represent minimising nodes. Alpha is initialized to negative infinity and Beta is initialized to positive infinity. When performing the Alpha-Beta Pruning with a MiniMax Algorithm, the following steps will be performed:

The algorithm follows the same steps as for the minimax until node K is reached. At this point $\alpha_A = 6$, $\alpha_D = 6$ and $\beta_D =$ Infinity. Next Node K has a value of 2, so $\beta_D = 2 < \alpha_D$ indicating it can be pruned. This is because 2 is less than α_D which is 6 and so regardless of the values of L and M. D will never be larger than 2 so A will chose C over D.

As shown in the above example the alpha beta pruning algorithm the average number of moves evaluated at each node is reduced. Reducing the number of moves evaluated at each node is more beneficial than reducing the number of steps as it reduces the

size of the only exponent in the equation that calculates the number of game states evaluated. Thus best case scenario sees the Alpha Beta pruning algorithm pick the same move as the MiniMax algorithm while only checking $O(b^{m/2})$ nodes (reduced from $O(b^m)$). This effectively allows alpha beta pruning to search a truncated game tree that is twice as deep as the MiniMax algorithm can search in an equal amount of time [21]. As such an additional move picking algorithm has been developed that augments our original MiniMax algorithm with alpha-beta pruning. It is hoped that this algorithm will perform better than the standard MiniMax algorithm. The algorithm is passed the current board state, the depth of termination, alpha equal to Integer.MIN and beta equal to Integer.MAX. It then searches as the game tree recursively on its children passing child game state, depth -1 and passing the updated α and β to its children. It only has one variable that can be changed by the user and that is "depth", as for MiniMax. Increasing this variable again allows the player to search more of the game tree at the cost of increased computational expense and longer running times.

3.3 Improvements to Alpha-beta Pruning

In addition to improving the algorithm from the standard MiniMax algorithm by using alpha and beta variables other alterations were made in order to further progress towards improving upon the standard MiniMax algorithm. These alterations build upon the gains made from having augmented the standard MiniMax algorithm with alpha-beta pruning by acknowledging the fact that the effectiveness of alpha-beta pruning is highly dependent on the order in which states are examined with alpha-beta pruning. It is made much more effective if better paths are evaluated first [21]. Obviously the move ordering cannot be done perfectly, because it would imply knowing what the best moves are and this knowledge could be used to play a perfect game. However, if this augmentation is done well, even if not optimally, then [21] estimate that the augmented algorithm does examine only $O(b^{m/2})$ nodes to pick a move, instead of $O(b^m)$ for MiniMax. This allows

the new more efficient augmented algorithm to solve a tree roughly twice as deep as MiniMax in the same amount of time. If successors are examined in a random order, the total number of nodes examined will be roughly $O(b^{3m/4})$, whereas a fairly simple ordering function gets you to within a factor of 2 of the best-case $O(b^{m/2})$ result [21]. Two algorithms have been developed that are expected to perform better than the alpha beta enhanced MiniMax algorithm.

The first is called RandomOrderAlphaBetaPlayer which works by randomly shuffling the order of the moves searched at each node in the game tree. This is in an effort to see a slight reduction in the number of nodes of the game tree checked overall when compared with our standard alpha-beta pruning algorithm which always evaluates the moves from each of the different game states in the same order. However, it is unclear at this stage if the overall effect of the change to the algorithm will result in a reduction in the number of nodes checked, this is because at some nodes in the game tree randomly shuffling the moves will allow more of the game tree to be pruned, while at other nodes the randomly shuffling of the moves will force more of the game tree to be searched. At each node of the game tree the list of legal moves available from the current game state is randomly permuted. The algorithm then searches the game tree going no further than the desired depth, using alpha-beta pruning in conjunction with the MiniMax algorithm. Again it only has one variable that can be changed by the user and that is "depth".

The second is a HeuristicOrderAlphaBetaPlayer which works by sorting the moves into descending order by utility if the node is a max node or into ascending order by sorting moves by the heuristic evaluation function if the node is a min node. This is in an attempt to have the best move evaluated first and thus allowing for more of the game tree to be pruned which in turn allows less nodes to be checked while still resulting in the high utility associated with having this algorithm stem from the MiniMax algorithm. This algorithm is passed the current board state, the depth of termination, alpha equal to Integer.MIN and beta equal to Integer.MAX. At each node of the game tree the list of

legal moves available from the current game state is sorted into descending order if the MAX player is evaluating a game state, or sorted into ascending order if the MIN player is evaluating a game state. Again it only has one variable that can be changed by the user and that is "depth".

Other improvements that can be made to the original MiniMax algorithm/alpha-beta pruning algorithms that are not implemented in this thesis but are acknowledged due to their effectiveness are:

- Killer Moves: which seeks to improve the alpha beta pruning by providing an order for the moves which sees a greater reduction in the size of the game tree that is searched by the regular Alpha Beta pruning algorithm. It does this by recording moves that in similar game states were shown to be optimal, and evaluating those first [21].

- Iterative Deepening: which uses depth first search but reorders the nodes it visits using breadth first search. This allows the algorithms to capitalize on breadth-first search's completeness while also taking advantage of the reduction in the size of memory needed being able to prune more of the game tree owing to the order in which it visits the nodes. As a result iterative deepening visits the same node in the higher levels of the game tree multiple times, it may seem wasteful, but this is only done in the higher level of the game tree and allows much more of the tree to be pruned at higher depths. However, this also results in the algorithm being robust against early termination due to time constraints [21].

- Aspiration Search: which seeks to increase the depth that can be searched to by the alpha beta pruning algorithm. The window of search as defined by the Alpha and Beta variables in the Alpha Beta pruning algorithm is narrowed. This allows more of the tree to be pruned by the algorithms, but unlike the MiniMax algorithm is not always guaranteed to pick the best move [21].

3.4 Experiment

Each of the algorithms either has 0 or 1 changeable variables. The RandomPlayer has no changeable variables while the MiniMaxPlayer, AlphaBetaPlayer, RandomOrderAlpha-BetaPlayer and HeuristicOrderAlphaBetaPlayer all have a changeable variable "depth" which operates as described above. The aim is to ascertain that all players are picking optimal moves, while testing their relative performance in terms of nodes checked and time taken.

3.4.1 Method

Experiment 1 - In the first test each of the algorithms will compute the move that they think is best for a list of 1000 game states. During the course of this test information will be collected on the average number of nodes of the game tree visited for each move made by each algorithm, the average time taken to pick each move for each algorithm and check the algorithms pick the same move in each case (except the RandomPlayer). This data will indicate which algorithms are the best for use in later experiments as determined by which has the shortest running time, and searches the smallest number of nodes of the game tree, but still picks a move with optimal utility.

Experiment 2 - In the second test the best algorithm chosen from the first experiment will play itself at varying depths from 1 to as high as the machine can compute in a reasonable amount of time. This data will allow determination of the value for the depth that gives the best performance in terms of the trade-off between quality of move and increased running time.

3.4.2 Expected Result

Experiment 1 - Table 3.1 shows how expected results for the experiments described in section 3.4.1 would turn out showing the expected relative number of nodes checked,

running times and utilities of the move picked by each of the algorithms that will be compared:

	Average number of nodes checked	Average running time	Relative utilities
RandomPlayer	Lowest	Lowest	Low
MiniMaxPlayer	Highest	Highest	High
AlphaBetaPlayer	High	High	High
RandomOrderAlphaBetaPlayer	Medium	High	High
HeuristicOrderAlphaBetaPlayer	Low	Medium	High

Table 3.1: Expected results for experiment 1(a) section 5.5.2

Note that the utilities of the moves picked should be the same, up to the small random weight r, for all of the algorithms except for the RandomPlayer. The small random weighting has been made small enough such that it does not influence the picking of a move except in situations where the moves would have otherwise had exactly the same utility. By preventing the same move being picked repeatedly it also prevents the same game being played repeatedly.

Experiment 2 - With regards the optimum depth intuition suggests that the values such as 1, 2 and perhaps 3 wouldn't be sufficiently big as they do not allow much of the game tree to be searched (allowing 8, 64 and 512 nodes of the game to searched respectively). Whereas values such as 9, 10 and 11 would be too big because the number of nodes checked grows at an exponential rate and thus depths of 9. 10 and 11 would check 134217728, 1073741824 and 8589934592 nodes of the game tree respectively, and it is expected that such a high number of nodes would be too computationally intensive for the machine available to be able to compute. Thus a depth between 4 and 8 ought

to give reasonable moves in a reasonable amount of time.

3.5 Results

Experiment 1 - Each of the algorithms except the random player produced moves with same utility (ignoring differences <= 0.1 which are due to the small random weighting r) which was the expected behaviour. This confirms that all the algorithms are picking optimal moves as expected. Fig 3.3 shows a comparison of the number of nodes checked by each of the algorithms. The numbers on the x axis represent the different algorithms:

- Numbers 1, 6, 11, 16, 21 and 26 represent the MiniMaxPlayer at depths 1, 2, 3, 4, 5 and 6 respectively.
- Numbers 2, 7, 12, 17, 22 and 27 represent the AlphaBetaPlayer at depths 1, 2, 3, 4, 5 and 6 respectively.
- Numbers 3, 8, 13, 18, 23 and 28 represent the RandomOrderAlphaBetaPlayer at depths 1, 2, 3, 4, 5 and 6 respectively.
- Numbers 4, 9, 14, 19, 24 and 29 represent the HeuristicOrderAlphaBetaPlayer at depths 1, 2, 3, 4, 5 and 6 respectively.

The y axis shows the average number of game states checked for each algorithm.

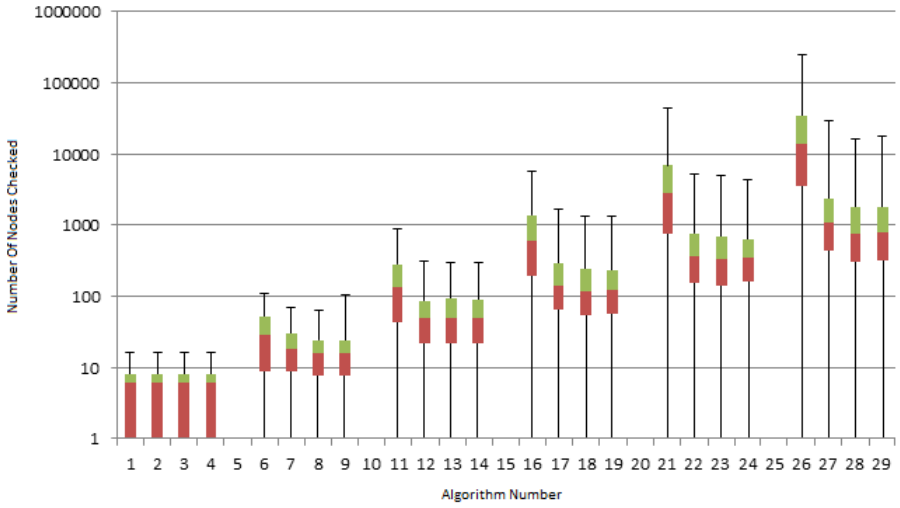


Figure 3.3: Average Number Of Nodes Checked Graph

Overall the results show a big reduction in the number of nodes checked when comparing the MiniMaxPlayer with the AlphaBetaPlayer, and again when comparing the AlphaBetaPlayer with the RandomOrderAlphaBetaPlayer and HeuristicOrderAlphaBetaPlayers. Upon closer examination it becomes apparent that there was not much difference in the performance of the RandomOrderAlphaBetaPlayer and the HeuristicOrderAlphaBetaPlayer although it seems that the HeuristicOrderAlphaBetaPlayer performed the best out of the algorithms compared in Experiment 1 because on average it checked the fewest number of games states while still picking a move of suitably high utility. A comparison of each of the running times of the algorithms shows the expected trends. Note that there were some game states with attacking moves available, which were forced. Hence each algorithm occasionally returned a move after checking 0 nodes. The results obtained from the experiment agree with those predicted in 3.1 with the MiniMaxPlayer picking moves with higher relative utilities but performing poorly in terms of average number of nodes checked and average running time. The AlphaBetaPlayer, RandomOrderAlphaBetaPlayer and HeuristicOrderAlphaBetaPlayer algorithms performed bet-

ter in all the areas we measured, with the HeuritsOrderAlphaBetaPlayer observed to be picking moves with high relative utilities, while checking the lowest number of nodes in the fastest time thus performing the best out of all the algorithms.

Experiment 2 - 3.4 shows the number of white wins resulting from the HeuristicOrder-AlphaBetaPlayer algorithm playing itself at depths ranging from 1 to 6.

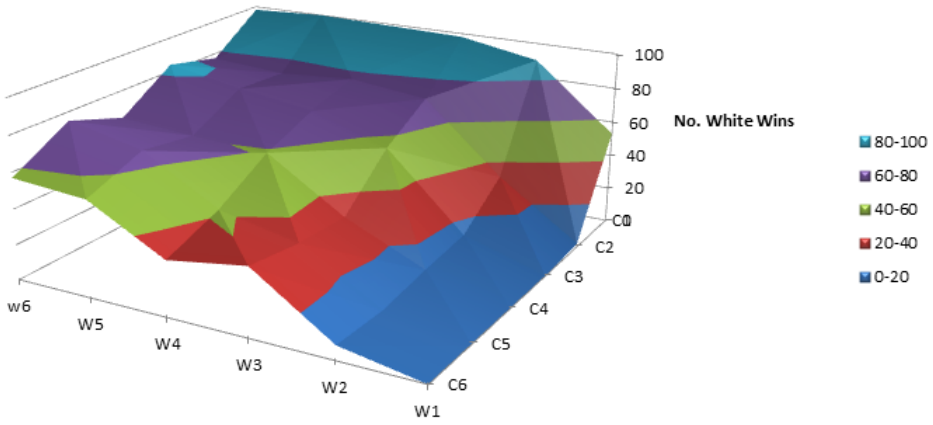


Figure 3.4: Heat Map showing white wins out of 100 games for varying depth of search for the white and black players. Point Wi Cj gives the number of wins when player W searches to depth I and player C searches to depth j

The general shape of the chart indicates that greater depth leads to a higher chance of winning, and reveals that there seems to be very little benefit to a player being allowed to have the first move in a game of draughts. Deeper analysis of the charts illustrates that the value for the depth that gives the best performance in terms of the trade-off between quality of move and increased running time is depth 4. Depth 6 search take on average 5 times as long but still only beat depth 4 searches 65% of the time.

4 Ant Colony Optimization

4.1 A Brief History of Ant Colony Optimization

Ant Colony Optimization algorithms are probabilistic algorithms which simulate foraging behaviour of ants to find good paths in graphs (or in our case good moves in a game tree). Ant Colony Optimization was first introduced as a concept in the PhD thesis titled 'Optimization, Learning and Natural Algorithms' published in 1992 by Marco Dorigo [8]. It was the first algorithm that utilised the behaviour of ants in the wild looking for the shortest path between two geographical points (the ant colony and the ants' source of food) as the mechanism for finding the shortest path between two nodes of a graph. Since then Marco Dorigo has been at the forefront of refining the Ant Colony Optimisation algorithm launching the first ant colony algorithm conference in 1998, as well as developing Ant Colony Optimisation algorithms for use on a wider class of problems such as the travelling Salesman Problem [9]. Dorgio's interest in Ant Colony Optimisations is shared by many other researchers interested in Artificial Intelligence whom have sought to further its study in games by using it in a variety of interesting ways including a MAX-MIN Ant System proposed by Hoos and Stützle in 1996 [30] and to play Chess [10].

4.1.1 Ant colonies in nature

In nature, ants begin their search for food by randomly traversing their environment. Upon finding such a food source the ant then returns to its colony while simultaneously

laying down a trail of pheromones thus creating a chemical link between the two points that can be followed by ants searching for food. Later when another ant who is searching for food finds the trail of pheromones they are less likely to continue randomly traversing their environment deviating from the path with the trail of pheromones on it to find new paths, and instead are more likely to use the persisting trail of pheromones as a guide and if they successfully follow it to a food source then further reinforce it by their own trail of pheromones. With an increased strength in the pheromones leading from the ant colony to the food source any ant who finds it is even more likely to follow it than the first two ants. However, these trails of pheromones do not build up indefinitely but instead evaporate over time. This evaporation over time results in an emphasis on shorter paths being reinforced as the best routes to take. This is because in the amount of time it takes to lay pheromone on a long path, more pheromone can be deposited on a shorter path because it takes less time to traverse. The evaporation over time also means that the route taken by the ant is not decided as soon as the first pheromone trail is laid. Instead bad routes see their pheromone trail diminish such that they are not taken in favour of more optimal routes. Thus ants in nature find optimal paths from their colonies to food sources in the surrounding environment [28].

4.1.2 Ant colonies as a model in computing

As previously described ants in nature can be observed to find the optimal path between two relatively close geographical points. This behaviour can be used in computer science by finding the shortest path between points in a graph, and by extension to solve more complex problem such as the Travelling Salesman Problem by having the ants leave the colony (start node) of a graph and find a route to the food source (start node again) passing through each of the other vertices of the graph exactly once. The Ant Colony Optimisation can try to do this optimally by having the ants explore each possible solution in the search space of the problem as they would search each possible path to

a food source in the environment surrounding their ant colony. In practice the details between specific Ant Colony Optimisation algorithms can differ greatly but one thing they all have in common is that they exchange information between ants via the environment by laying pheromone deposits.

Path selection by an ant is done probabilistically based on two key parameters that determine the transitions between nodes: the desirability (or attractiveness), η_{ij} , and pheromone level, τ_{ij} of the edge between the two states i and j. η_{ij} is usually represented by a predetermined heuristic, and therefore indicates an a priori fitness of the path. Whereas, τ_{ij} indicated the past success of the move, and therefore represents a posteriori fitness of the path. The update for τ_{ij} takes place once all the ants have finished a forage. Other parameters factored into the decision made by the ant include α and β which are user-defined parameters that determine how much influence should be given to the trail strength and desirability respectively, and M which is the set of all legal moves that can be made from state i. Given those parameters, the probability p_{ij} of selecting a path between states i and j is given by [28]:

$$p_{ij} = \frac{(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}{\sum_{k \in M}(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}$$

Pheromones are deposited by an ant once they have finished a forage through the state space. The trails are updated using the parameters $\triangle\tau_{ij}$ which is the cumulative accumulation of pheromone by each ant that has passed between i and j, and ρ which determines the rate at which the existing amount of pheromone decreases using the following equation [28]:

$$\tau_{ij}(t) = \rho\tau_{ij}(t-1) + \triangle\tau_{ij}$$

$$\text{where } \triangle\tau_{ij} = \sum_{ants\, using\, transition\, ij} heuristic\, score\, of\, final\, state\, reached$$

A pseudocode algorithm for an ant colony optimisation algorithm adapted from [28] is given in algorithm 4.1:

Algorithm 4.1 Ant Colony Optimization pseudocode example

```
Initialise τ and η parameters
while Terminal condition is not met do
    for all Ant a in AntColony do
        while solution is not complete do
            select a transition t probabilistically
        end while
    end for
    for all Trails t_ij made by all Ants do
        update τ_ij(t) ← ρτ_ij(t − 1) + △τ_ij
    end for
end while
```

4.2 Experiment

The Ant Colony Optimisation algorithm has a number of changeable variables. These include variables that control the trade-off between running time and increased move utility: "depth" determines the maximum layer in the game tree searched, "totalNumberOfAnts" is set to determine the number of ants in the ant colony, "totalForages" is set to determine the total number of forages made by each ant. The algorithm also has several changeable variables that affect the learning rate of the algorithm: a variable α determines strength of the influence of the pheromones deposited by the ants, a variable β determines the strength of the influence of the desirability of a route determined by the previously described scoring heuristic, a variable ρ determines by how much the pheromone trail deposited by the ants evaporates per forage, and a variable ν determines how significant the length of a path is in assessing its viability.

Experiments will be carried out using the Ant Colony Optimisation algorithm to achieve two main goals:

1. To use the ant colony optimisation algorithm to find the best values to assign to each of the changeable variables that influences the working of the implemented algorithms such that they are individually optimised.

2. To see how it performs compared with the best of the alpha beta pruning algorithms found in the previous chapter, and to ensure that the ant colony optimisation algorithm functions as expected

4.2.1 Method

Experiment 1 - Tests will be conducted to try and identify the best values to assign to each of the variables during the course of the subsequent tests. This will be done by running the algorithm with a range of values for each of the changeable variables and seeing which variables increase the algorithms performance against the HeuristicOrderAlphaBetaPlayer. For the 'totalNumberOfAnts' and 'totalForages' variables these values will range from the highest possible number that still allows the algorithms to pick a move within a reasonable time frame e.g. 1 minute per move, to an arbitrarily low number such as 5 which isn't expected to be a value that offers a good performance but is instead chosen as a lower bound in the interest of scientific integrity by ensuring as many values are considered as possible. The values α, β, ρ and ν will be tested using a range of values matching those suggested by other academic publications which are presented in table 4.1.

	Variables					
Experiment Number	totalNumberOfAnts	totalForages	α	β	ρ	ν
1	a(max)	b	c	d	e	f
2	a-75%)	b	c	d	e	f
3	a(50%)	b	c	d	e	f
4	a(25%)	b	c	d	e	f
5	a(min)	b	c	d	e	f
6	a-best	b(max)	c	d	e	f
7	a-best	b(75%)	c	d	e	f
8	a-best	b(50%)	c	d	e	f
9	a-best	b(25%)	c	d	e	f
10	a-best	b-(min)	c	d	e	f
11	a-best	b-best	c(max)	d	e	f
12	a-best	b-best	c(50%)	d	e	f
13	a-best	b-best	c(min)	d	e	f
14	a-best	b-best	c-best	d(max)	d	e
15	a-best	b-best	c-best	d(50%)	d	e
16	a-best	b-best	c-best	d(min)	d	e
17	a-best	b-best	c-best	d-best	e(max)	e
18	a-best	b-best	c-best	d-best	e(50%)	e
19	a-best	b-best	c-best	d-best	e(min)	e
20	a-best	b-best	c-best	d-best	e-best	f(max)
21	a-best	b-best	c-best	d-best	e-best	f(50%)
22	a-best	b-best	c-best	d-best	e-best	f(min)

Table 4.1: Ant Colony Optimisation Variable Values

Experiment 2 - The Ant Colony Player will compute the move that it perceives is best for a list of 1000 game states. The depth will be set to match that used by the best algorithm as found in the previous chapter (HeuristicOrderAlphaBetaPlayer). Throughout this test information will be collected on the average number of nodes of the game tree that are visited, the average time taken to pick each move, and the average utility of the move picked by the AntColonyOptimisationPlayer. When this data is compared the best algorithm will be determined.

Experiment 3 - Finally the best algorithm from the previous chapter HeuristicOrder-

AlphaBetaPlayer and the AntColonyPlayer play against each other for 100 games per round. Each round will see the algorithm chosen from the previous chapter have its depth remain unchanged while the depth of the AntColonyPlayer increases by 1. This will indicate the depth at which the ant algorithm starts wining as many games as the best algorithm from the previous chapter. Using this information it will be possible to judge how the ant algorithm fares against the best algorithm chosen from the previous chapter and be able to recommend which one is better to use.

4.2.2 Expected Result

Experiment 1 - I would expect the changeable variables to converge to values close to those suggested by [28] in which Ant Colony Optimisation is also applied to game tree search. These values are presented in table 4.2.

totalNumberOfAnts	totalForages	Alpha	Beta	ROH	NU
40	200	0.6	0.8	0.8	1

Table 4.2: Recommended Ant Colony Optimization Variables

Experiment 2 - For the second experiment where the Ant Colony Optimisation based algorithm is ran at the same depth as the Alpha-beta pruning based algorithm in the previous chapter I would expect the Alpha-beta Pruning Player to outperform the Ant Colony Optimisation Player. This prediction assumes that the ant algorithm will fail to function as well as the HeuristicOrderAlphaBetaPlayer when given the same value for depth. This is owing to the fact that the Alpha Beta pruning algorithm will pick the optimal move, whereas the Ant Colony Optimization algorithm is not guaranteed to do the same as it utilises a heuristic.

Experiment 3 - For the third experiment it is expected running the Ant Colony Optimisation based algorithm at greater depth than the Alpha-beta pruning based algorithm will result in the AntColonyPlayer outperforming the AlphaBetaPlayer but whether this is at too great a cost to the running time cannot be known at this stage.

4.3 Results

Unfortunately the Ant Colony Optimisation algorithm's performance meant that in comparison with the HeuristicOrderAlphaBeta algorithm it performed very poorly. This is because the Ant Colony Optimization algorithm could not handle the sufficiently high number of ants and forages required to explore enough of the game tree. The following is an analysis of the limited results collected when the AntColonyOptimisationPlayer and the HeuristicOrderAlphaBetaPlayer played each other at depths ranging from 1-10.

With regards Experiment 1, the Ant Colony Optimisation algorithm didn't function rapidly enough to allow us to start on the table of tests that were planned to be conducted. The planned experiments required that the Ant Colony Optimisation algorithm could facilitate having a minimum of 20 ants make a total of 100 forages and a maximum of 80 ants making a total of 400 forages. However, at its peak the Ant Colony Optimisation only managed to have 15 ants make a total of 15 forages for a total of 1000 games for depths equal to 1 through 10. The aim was to get hundreds/thousands of ant and forages for a similar number of games over the same time period to produce an extensive set of results with which the HeuristicOrderAlphaBetaPlayer can be compared. This experiment was abandoned due to the extensive expanse of time it would have needed to run.

With regards Experiment 2, graph 4.1 shows the utility of the generated move under the HeuristicOrderAlphaBetaPlayer (x axis) and the AntColonyOptimisationPlayer (y axis) for each of the 1000 test game-states when both algorithms played to depth 4.

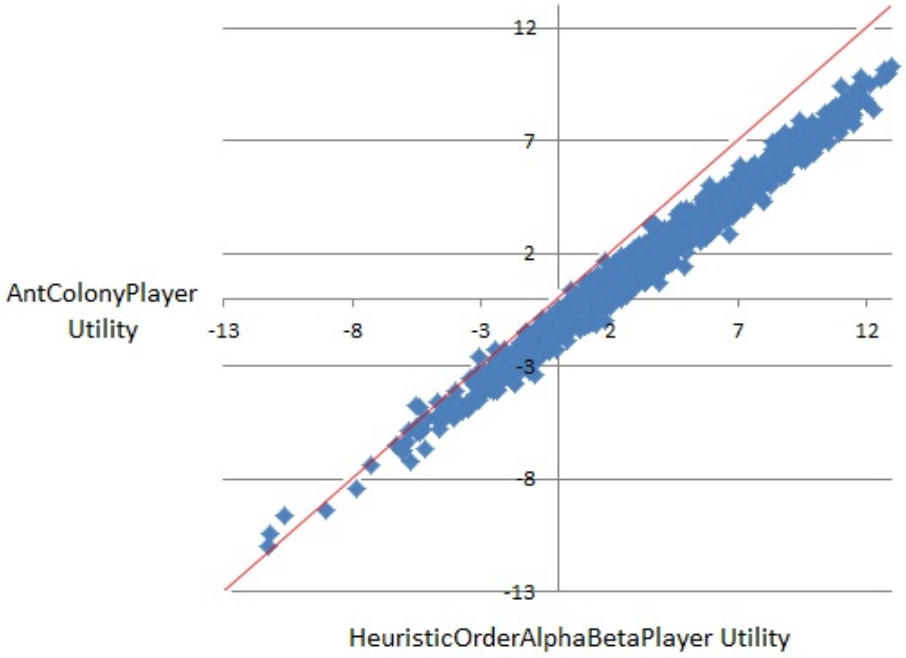


Figure 4.1: AntColonyOptimisationPlayer Versus HeuristicOrderAlphaBetaPlayer each playing to depth 4

The fact there is a linear relationship shows that it is picking better moves when the optimal is better than when the optimal is worse, this can be attributed to the fact that the 1000 game boards for which these two algorithms picked moves represents a fair spread of moves over the course of a game of draughts. The random test game-states were made beginning with a new game setup for each of the 1000 board states. The random player then played 2 random moves for every 40 board states e.g. the first 40 random board states were made by having the random player play 2 moves (one for the coloured player and one for the white player). The next set of 40 test game-states were made by having the random player play 4 moves etc. This continued until the last set of

random games states the random payer was playing 50 moves before the algorithms were allowed to pick the move they thought was best. Additionally the above graph shows some bias towards positive utilities which can be attributed to each players trying to pick the best moves they can i.e. given the choice between -1, -6 and 2 they will pick 2. There are also a few points above the line which can be attributed to the small random weighting 'r' which is added to each score. The graph shows that even at depth 4 there is a significant difference between the two algorithms. This is indicated by the graphs cloud of points being below the red x=y line which shows that the HeuristicOrderAlphaBetaPlayer algorithm is picking moves with noticeably higher utilities and is thus outperforming the Ant Colony Optimisation algorithm which is not picking optimal moves.

Finally with regards Experiment 3, graph 4.2 shows the average number of seconds taken for each of the algorithms HeuristicOrderAlphaBetaPlayer (shown in red) and the AntColonyOptimisationPlayer (shown in blue) to pick at move at each of the depths 1 through 10.

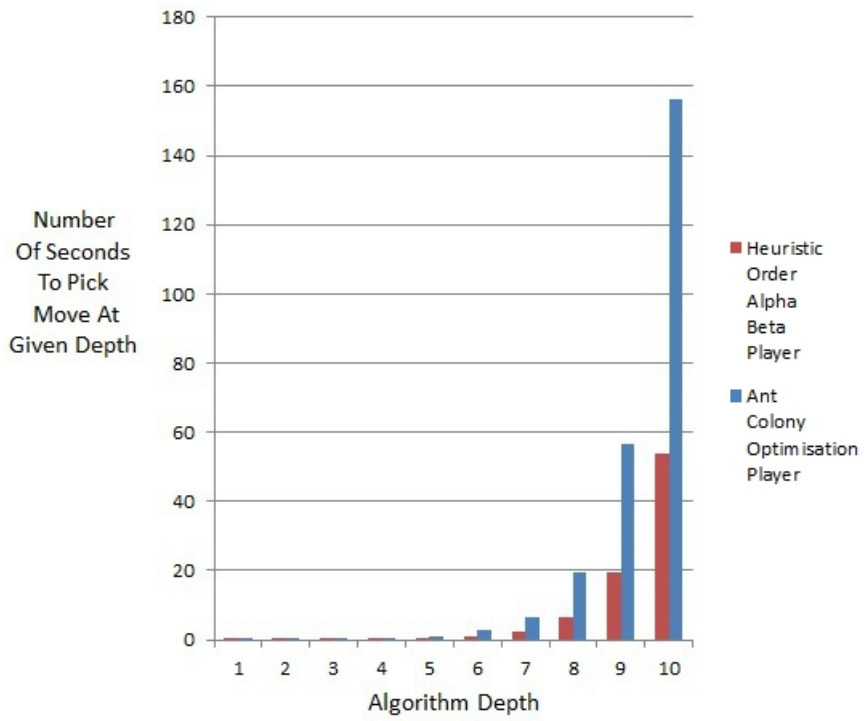


Figure 4.2: AntColonyOptimisationPlayer Versus HeuristicOrderAlphaBetaPlayer Time Comparison

The graph shows that the Ant Colony Optimisation algorithm takes a lot more time to pick a move when compared with the HeuristicOrderAlphaBeta algorithm especially at higher depths, taking between 2 and 4 times longer for it to pick a move. Thus despite the Ant Colony Optimisation algorithm being expected to run slower than the HeuristicOrderAlphaBeta algorithm managing to run at depth 10, the fact that it ran so slowly resulted in the experiment being abandoned.

The biggest problem with the algorithm was the way the program stored and managed the list of all of the game states visited by the ants. Once an ant has finished traversing a path the list of states previously visited by other ants would be checked for each game

state visited by that ant in order to lay pheromone. For example if 15 ants are released onto the game tree to a depth of d for 15 forages then the number of times the list of game states visited by previous ants would be checked would be equal to 15×15×d, and when you consider that the size of the list of game states stored by the program is also ≈ 15×15×d this amounts to ≈ $50000d^2$ individual checks. It is easy to see how this could be very detrimental to the running time of the algorithm.

Other problems that would have caused the algorithm to run slowly stem from the fact that Java was used and that deep copies of the game states were made, whereas if C had been chosen then pointers could have been used. Doing this in conjunction with only saving one copy of repeated board states would have reduced the number of game states that needed searching through and the amount of memory used by the algorithm thus improving its performance.

The Ant Colony Optimisation algorithm has been used successfully on the Travelling Salesman problem. This is a problem where a graph made of vertices representing cities, weighted edges representing the roads that connect them, and the weight being the cost of traversing these roads (given as time, distance or cost of fuel etc) is inputted. The problem is an optimisation one in which the solution takes the form of a cycle that visits each city exactly once with the smallest possible total weighting. The game trees in the draughts problem and the weighted graphs in the travelling salesman problem both consist of a huge number of edges. However, the Ant Colony optimisation solution proposed for that travelling salesman problem requires only n^2 pheromone levels to store and track the non-zero pheromone trails even though state space is size n!=O(2^n). As such a new model could be proposed for future research based upon the vertices of the graph representing game states and the edges being represented by moves, the difference being that now ants lay pheromone on a good move from the current board state instead of on the combines move and game state. This has the advantage of reducing the number of moves analysed from each board state to 4×32 at the cost of knowing no other information about the rest

of the game. It is expected that utilising this newly conceptualised graph would allow a lot more ants to be released for a higher number of forages.

5 Genetic Algorithm

5.1 A Brief History of Genetic Algorithms

Genetic algorithms are based on the notion of survival of the fittest and seek to simulate the process of natural selection. They are useful in problem domains that have complex fitness landscapes. The concept was introduced by Nils Barricelli [3] when he produced a program using a genetic algorithm that showed an improvement in the ability to play a simple game of Tac Tix published by Gardner [11]. The concept of genetic algorithms was developed further as a field by Ingo Rechenberg and Hans-Paul Schwefel whom applied the techniques to optimise the shape of wings and other shapes to reduce drag [26]. The First International Conference on Genetic Algorithms was held in Pittsburgh, Pennsylvania in 1985 [1]. Since then there has been much progress in the field with genetic algorithms finding application in computational science, engineering, economics, chemistry, mathematics, physics and other fields this include such work done by John Koza whom in 1992 used genetic algorithm to evolve programs to perform certain tasks[13].

5.1.1 Genetics in biology

Survival of the fittest is a process that occurs in nature by which natural selection sees the members of a population develop traits to better suit their environment resulting in a fitter population. This process comes about via two mechanisms: random variations in inheritance and greater breeding of fitter members of the population. The first of these is brought about in two ways: the recombination of the members of the population to

produce children and mutation. The recombination of the members occurs when two members of the population randomly pair up and produce a child. An example of this can be shown if two parent binary strings of the same length are considered A = 000000 and B = 111111. If the two parent strings were used to produce a child using the first half string A and the second half of string B then a child binary string C = 000111 will have been produced which poses a mixture of the phenotype of each of its parents (0's and 1's in our example). The second way is the mutation of randomly selected children. An example of this would be to take the child binary string C = 000111 that was just created and to randomly select a position in the string at which to flip a bit. If the randomly selected position was 2 the bit at that position would be flipped resulting in the production of the mutated child binary string = 010111. The process of mutation sees the randomly selected child altered in a way that randomly introduces new characteristics into the population allowing for greater diversity of the population members. The second of these is brought about by ensuring that fitter members of the population which are evaluated as such according to a fitness heuristic have a higher chance of being chosen to become parents. This ensures that better solutions have a higher chance of influencing the evolution of the solution such that the population will converge towards a more optimal solution. This process of evolution can lead to adaptation in the members of the population and can see and improvement in the species as a result. Genetic algorithms attempt to replicate this effect to solve optimization problems by evolving towards better solutions.

5.1.2 Genetics as a model in computing

Genetic algorithms replicate the effects of natural selection by starting from a population of randomly generated members which can consist of hundreds of thousands of members each one of which is a solution to the problem. This allows a broad range of solutions to be considered by the algorithm which increases the likelihood of finding the optimal

solution. At each generation the members of the population are evaluated according to a fitness heuristic. A pair of members from the population is then selected such that members that are evaluated to be fitter have a higher chance of being chosen to become parents of a new member solution. The parents are then recombined to produce a child member solution which shares parts of its solution with both of its relatively fit parents. This child may then be randomly selected to be randomly mutated such that its solution is randomly altered in an effort to increase the search space of solutions considered. The process of producing children from selected pairs of members happens several times per generation to produce a new population whose average fitness will have increased since only the fittest members from the previous generation were selected to be parents. The algorithm terminates when a predefined number of generations has been reached, or when the fitness of the population exceeds a predefined level.

5.2 Experiment

The Genetic Algorithm has the following changeable variables, a variable: "populationSize" determines the size of the population used by the genetic algorithm, a variable "numberOfGenerations" determines the number of generations though which the population is bred and mutated, and a variable "learningRate" determines the speed at which the population is allowed to converge to the ideal population.

The genetic scoring algorithm is used to find the best values for the scoring matrix B. Recall that B is an 8×8 matrix that contains a value for each board square that is representative of how desirable it is to occupy that square with a piece, and is used in the heuristic evaluation function together with the matrix which stores the location of the white and coloured pieces on the board to calculate the value of a given board state.

Owing to the nature of the 2 player game creating a situation where there is always a winner and a loser (the combination of conditions detailing the Draughts section creating a situation where it is impossible for there to be a draw between the two players) the

number of the population at each generation is halved as only the winners are kept. The population is then built up again to its original size by creating children and mutated versions of the children with the following percentages: 50% winners from the previous population, 25% children of the previous population and 25% mutated children of the previous population.

In the experiments breeding occurs by selecting two random parents from the winners of the last generation, with parents that are evaluated as fitter by the heuristic evaluation function having a higher probability of being picked, and then a random number is generated between 0 and 7. This random number specifies which of the 8 columns in the matrix becomes the point at which the two parent matrices are joined to make the new matrix e.g. if the two parents where:

1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2

and

1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2

Supposing that the random number was 2 then this would be the resulting child matrix:

	1		2		2		2
1		1		2		2	
	1		2		2		2
1		1		2		2	
	1		2		2		2
1		1		2		2	
	1		2		2		2
1		1		2		2	

In the experiments each of the children is generated from a child as follows. The values in a copy of the 8×8 matrix of the child have a 10% chance to be mutated to a different value. When it is decided that a value in the child will be mutated, a random bit is generated to determine whether the child's value will be increased or decreased. Once this is decided the child's value is changed thusly:

mutated value = ((child's value)×(1 ± (learning rate));

The learning rate decreases as the number of generation increases. This has the effect of producing plenty of randomness in the beginning, but allows the algorithm to converge on a local optimum later on.

5.2.1 Method

The best values for the scoring Matrix B will be found by conducting a series of tests:

Experiment 1 determines the best parameters - The following tests will be run by initialising a random population and then proceeding to breed and mutate these matrices by playing 100 games per generation with the following settings for the other variables

to determine which set of variables gives the best scoring matrix. This will enable the identification of the best variable settings to use:

Test	Population	Number Of Generations	Initial Learning rate
1	1000	10	0.25
2	1000	10	0.50
3	1000	10	0.75
4	100	100	0.25
5	100	100	0.50
6	100	100	0.75
7	10	1000	0.25
8	10	1000	0.50
9	10	1000	0.75

Table 5.1: Experiment 1 genetic algorithm settings

Experiment 2 uses good parameters to get good Matrix B - A test will be run by initialising a random population and then proceeding to breed and mutate these matrices by playing 100 games per generation using the parameters which fared best in experiment 1. This will give the genetic algorithm the best chance to produce an altered scoring Matrix B that will improve how much the artificial intelligence algorithm wins.

5.2.2 Expected Results

It is expected that the best settings for the genetic algorithm would be ones that had both a large population and a large number of generations e.g. population = 100 and number of generations = 100. This is because if you have a large population but a small number of generations then you will have little to no convergence to a better solution than the parents in the population that you started with. However, you might 'get lucky' and have a parent in your population that resembles the best solution or close to one. Whereas if you have a small population but a large number of generations you will have some convergence to a better solution than the parents in the population you initially

started with, but you are limiting the chances that a child resembling the best solution or one close to it will exist in the population.

The best learning rate is one that allows the population to change fast enough to produce a child that is close to the best solution but which is slow enough to allow the child to have an influence of changing the population so in this case the best learning rate expected to be 0.5.

5.3 Results

The results of experiment 2 showed that test 2 was the best. These settings were then used to run with more generations to get B. After having run the Genetic Algorithm the parameters described above in test 2 for 12000 generations the following matrix was obtained:

	1.0		1.0		0.9		1.0
0.8		0.8		0.7		0.7	
	0.6		0.6		0.5		0.6
0.6		0.6		0.5		0.4	
	0.5		0.4		0.5		0.6
0.4		0.3		0.5		0.3	
	0.2		0.3		0.2		0.4
0.3		0.2		0.3		0.4	

Table 5.2: Alternative scoring matrix produced by genetic algorithm

	1.0		1		0.9		1.0
0.8		0.8		0.7		0.7	
	0.6		0.6		0.5		0.6
0.6		0.6		0.5		0.4	
	0.5		0.4		0.5		0.6
0.4		0.3		0.5		0.3	
	0.2		0.3		0.2		0.4
0.3		0.2		0.3		0.4	

This matrix was then used by the HeuristicOrderAlphaBetaPlayer to play a total of 100 games against another HeuristicOrderAlphaBetaPlayer using a matrix compose entirely of 1's. The HeuristicOrderAlphaBetaPlayer using the matrix altered by the genetic algorithm won 64 % of the time. This is up approximately 14% from when both HerusiticOrderAlphaBetaPlayers played each other were using a matrix of 1's and won 50% of the time. A statistical analysis can be used to test if this increased probability of winning is statistically valid by testing against the Null Hypothesis that each player wins with probability $\frac{1}{2}$. Under this hypothesis the sample number of wins is distributed binomially (100, 0.5). This is approximately normal with mean 50 and standard deviation 5. Our observed data, 64 is $\frac{14}{5} \approx 2.8$ standard deviations above the mean. This can therefore be calculated to occur with probability $<1\%$ thus allowing us to dismiss the null hypothesis with 99% confidence. It can be concluded that the scoring matrix proposed by the genetic algorithm was the sole contributor to the increased performance of the player using it to win more games as both players were using the same move picking algorithms.

Furthermore the matrix altered by the genetic algorithm produced some interesting patterns. The first is that the value assigned to each of the board positions seems to increase from the players starting end of the board. The second is that the values assigned

to the squares around the edge of the board seem to be slightly higher than the values assigned to the squares in the centre of the board. Unfortunately it cannot be stated that to what degree each of the patterns seems to have influenced the player ability to win more games. However, it can be said that that this genetically altered matrix seems to suggest greater emphasis on moving the piece to the opponent's end of the board and trying to occupy pieces on the edge of the board over pieces in the centre of the board be proposed as winning strategies. This confirms the intuition that occupying a piece on the edge of the board is beneficial as it means that your piece can not be taken.

6 Game Implementation - Analysis Of Code

In this section details are given of the class structure, user interface design, system architecture, system design, the software tools, and important data structures and algorithms used to create our version of the board game Draughts and its various players.

6.1 Class Structure

It was decided to design the program in an Object Oriented way by grouping closely related sets of functions derived from the rules of draughts and also the various artificial intelligence algorithms that were implemented into classes. This resulted in the creation of 3 packages and 16 classes. Only 2 classes exist outside of a package, they are the 'Main' class and the 'Settings' class. The 'Main' class is called when the program is initialised. It contains a lot of the code auto generated by NetBeans to create the graphical user interface, along with code created by me such as individual button functionality and also the RandomPlayer. The 'Settings' class is used to store the game settings keeping them all on one place.

The 'draughts' package contains all of the classes that allow the program to play the game of draughts. These are the 'Draughts' class which is responsible for allowing the game to progress from state to state. The 'DraughtBoard' class contains all the information about the game at the present state, and the 'Scoring' class which provides

a separate class for the heuristic evaluation function which scores game states.

The 'players' package contains a total of 7 classes, this is one class for each of the AI player classes except for the random player whose implementation was too simple to justify the creation of a class dedicated to it, and the AntColonyOptimisationPlayer which was broken down into 3 classes. The 4 player classes are the 'MinMaxPlayer' class, 'AlphaBetaPlayer' player class, the 'RandomOrderAlphaBetaPlayer' player class and the 'HeusisticOrderAphaBetaPlayer' each of which picks a move using the algorithms they are named after to select moves. As previously mentioned the AntColonyOptomisationPlayer was broken down into three classes. These are the 'Ant' class which encapsulates an ant as an object, the 'AntPath' class which encapsulates a game state and a move as travelled by an ant for usage by the ant class, and the 'ACOPlayer' class which picks a move using an ant colony optimisation algorithm.

The 'testingFrameworks' package contains the classes that allow the program to persist data and display it in meaningful ways; this makes a total of 4 classes. These classes are the 'Database' class which enables the program to create, read and write to a database, the 'XML' class which enables the program to write the data from the database into xml form readable by Microsoft Excel , the 'GameDisplay' class which allows the reconstruction of games played in either a forwards or backwards direction using the data from the database, and the 'GenerationDisplay' class which displays the data produced by the genetic algorithm showing how the scoring matrix evolves through generations.

6.2 User Interface Design

Having designed the internal structure of the draughts playing program as described in the above section, the program's user interface was designed creating mock ups of how the graphical user interface should look. Owing to the proof of concept nature of the software being designed it was decided to make the various functions of the software easier to access at the cost of making it look aesthetically pleasing i.e. reducing the number of

tabs while increasing the number of buttons on those tabs. It was decided to use a tab system assigning tabs for each of the following 'Play Game', 'SQL Data', 'Excel Data' and 'Genetic Algorithm Play Back'. The 'Play Game' tab will be used to select the algorithm and depth searched to for each of the white and coloured players and start new games running. It was also used to watch games archived in the SQL database. The 'SQL Data' tab was used to facilitate data management containing buttons for resetting all of the data in the database, generating 100 random boards, displaying the data and a button for each of the experiments detailed in the previous chapters which generates the data for those experiments. The 'Excel Data' tab contains buttons for creating a Microsoft Excel readable xml document using the data in the database and another button which will open the xml document using Microsoft Excel. The final tab 'Genetic Algorithm Play Back' will be used to control the applet which displays the data produced by the genetic algorithm showing how the scoring matrix evolves through generations. Figure 6.1 shows how the program looked (with the Game play tab selected) :

Figure 6.1: GUI Template

6.3 System Architecture and System Design

The draughts program was developed using the incremental design model shown in figure 6.2 instead of the waterfall or the evolutionary models. This is due to the theoretical nature of this project requiring the algorithms to be changed or amended to ensure the

best techniques were being used and the algorithms were working properly. Therefore it made sense that the coding and testing sections were iterated over to ensure the program was of the highest quality. Whereas the waterfall model assumes that when each stage of the process is finished then it will never be returned to. This causes problems when research uncovers amendments to the algorithms that would result in improved performance but the coding stage has been completed never to be returned to and could in a worst case scenario result in the research undertaken being invalidated. In contrast the evolutionary model sees the whole life cycle of the program iterated over and not just the coding and testing stages. This would have proved impractical as it would have severely restricted the number of interactions that could have been done, while simultaneously loosing focus on the algorithms which are the focus of this research.

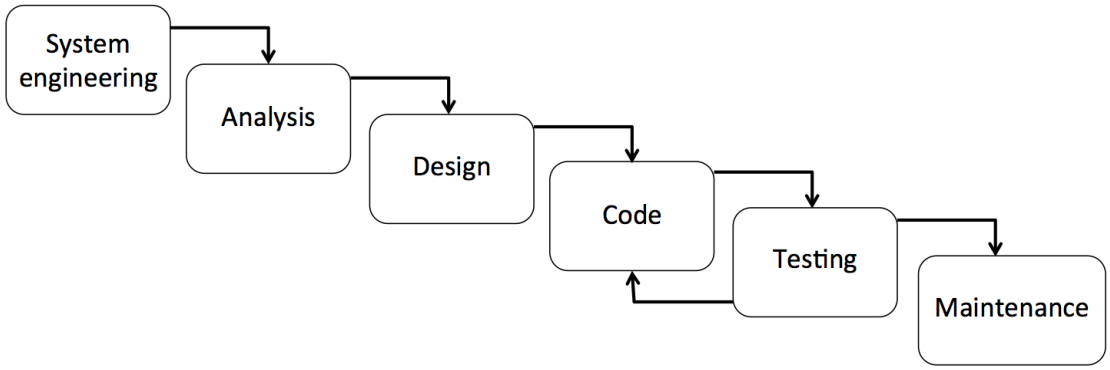


Figure 6.2: Incremental Design Model

The three-tier variant of the Multitier Architecture was used as the basis of our system architecture. The three-tier architecture subdivides the system into 3 tiers. These tiers are the presentation tier which handles the user's interaction with the program through the user interface, the logic tier, which handles data access and modification, and finally the data tier, which is where all the data in the system is persistently stored. The

architecture requires the separation of the packages and classes within each of these tiers stating that the presentation tier should not access data directly, nor should the data tier have any direct links with the graphical user interface. Instead the logic layer should be solely responsible for handling data transfer between the presentation layer and the data layer.

This style of system architecture was chosen because it complements the object oriented class structure described in the 'Class Structure' section of this thesis. The presentation layer includes the 'Main' class. The logic layer includes all the classes that are concerned with the rules of Draughts (e.g. the 'Draughts', 'DraughtBoard' and 'Scoring' classes), the AI players (e.g. 'MinMaxPlayer', 'AlphaBetaPlayer', 'RandomAlphaBetaPlayer', 'RandomOrderAlphaBetaPlayer', HeuristicOrderAlphaBetaPlayer, 'ACOPlayer', 'Ant' and 'AntPath' classes), the testing framework (e.g. 'Database', 'XML', 'GameDisplay' and 'GenerationDisplay') and some of the other classes (e.g. 'Settings'). Finally the data tier includes any files produced by the program in order to persistently store data e.g. database produced in the form of .db files.

An advantage of using this system architecture is that any changes made to the packages in any one of the layers will have a minimal effect on the packages in the surrounding layers. This is useful because the theoretical nature of the project required a lot of focus to be put on trying to get the best implantation of algorithms in the logic layer, and this architecture allows us to make frequent changes to the packages in the logic layer without having to worry about affecting the packages in the other two layers. However, a disadvantage of using this system architecture is that it doubles the communication points needed for data transfer as data has to pass from the data layer through the logic layer to the presentation layer, instead of passing it directly to the presentation layer. The system architecture that was chosen is illustrated in the figure 6.3:

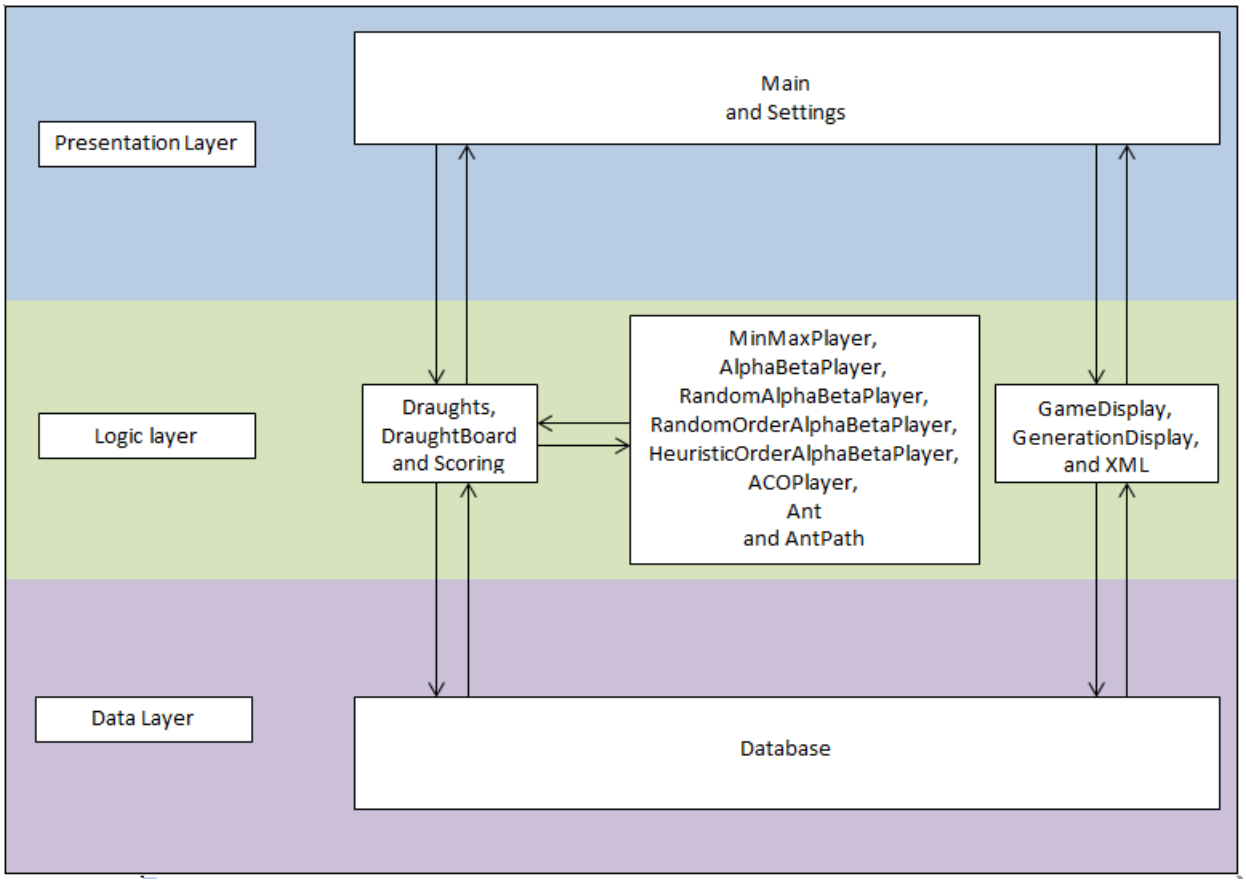


Figure 6.3: System Architecture Diagram

Other design choices made concerning the system architecture include:

1. Keeping the user interface as intuitive to use as possible.

2. Using a database. This was chosen instead of storing the data internally in the program because it meant that data could be persisted after the program was closed. It was felt that producing database files made more sense than choosing to persist the data via some other less practical methods such as sorting the data in one or more text files. Using this alternate method would have required a lot more

time be spent developing a mechanism for storing the data at the cost of reducing the time that could be spent developing the algorithms.

3. Utilising XML to produce Excel results.

6.4 Tools used

Throughout the course of this project the draughts program was developed using the following software:

6.4.0.1 Java

Java (Version 7, Update 10): A high-level programming language created by Sun Microsystems and now maintained by Oracle [20]. It is an object-oriented language similar to C++. Java was chosen in favour of other programming languages because [7]:

1. It eliminates language features that cause common programming errors (e.g. memory mismanagement). This meant that more focus was put on developing the algorithms than (in the case of memory mismanagement) spending time fixing common memory management issues such as memory that is allocated but has nothing pointing to it, and memory that has been allocated and is still referenced somewhere but that is no longer required.

2. It allows the utilisation of the Java Class Library which is a set of libraries that feature prewritten code that can be used within our software. Utilising pre-existing functions where appropriate time could be saved by avoiding coding every part of the proposed solution from scratch.

3. It is compiled into java bytecode that can be run by the java virtual machine which allows the draughts program to be run on computers that utilise a variety of different operating systems.

6.4.0.2 NetBeans

NetBeans IDE (Version 7.1): An Integrated Development Environments (IDE) that facilitates the writing of code in several languages and also compiles it. NetBeans was chosen in favour of other IDE's because [6]:

1. It supports a variety of plug-ins which can be used to do a variety of tasks such as profiling which facilitates the monitoring of the run time of methods used by the program. This provided valuable feedback enabled various algorithms implemented to be improved.

2. It facilitates easier creation and editing of Graphical User Interfaces.

6.4.0.3 SQLite/JDBC

SQLite (Version 3.7.12.1): SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite was chosen instead of other forms of persistence because [12]:

1. It is the most widely deployed SQL database engine in the world and thus has been tried and tested as being a reliable and helpful tool for software persistence.

2. It is contained in a small programming library allowing it to be bundled with the application but without contributing too much to its size. However, despite its small size it can still be used to handle the large amounts of data that will be collected over the course of the experiments.

3. It is been specifically designed to store data in a way that will prove useful to the format of this thesis unlike reading from and writing to text files e.g. 1 database file can store multiple tables of data whereas trying to construct a similar data structure in a text file would require either writing code to create multiple text

files, or try to partition one text file into several sections which would be time consuming.

7 Evaluation

The project was a success for the most part, permitting the completion of the majority of the objectives that were aspired towards at the beginning of this project. The game implementation chapter detailed the specific implementation of the game of draughts in the JAVA programming language used in this project, and the results in the previous chapters indicated that the game worked correctly as per the English variant of the rules of draughts.

The MiniMax chapter demonstrated the ability of the program to implement a variety of AI players. These AI players all offered a preferable alternative to both the naive brute force and random algorithms. Four out of the five proposed move-selection techniques were successfully tested in isolation showing that Minimax was worst in running time, and random was worst is game performance, and that the best of these was the HeurisitcOrderAlphaBeta algorithm. Unfortunately the Ant Colony Optimisation algorithm's performance was insufficient to obtain results that would permit a meaningful comparison between the Ant Colony Optimization algorithm and the HeuristicOrderAlphaBetaPlayer but this thesis has suggested alternative ways that could be implemented to do this in the future.

Finally the genetic algorithm was produced with the ability to augment the scoring heuristic without the benefit of external guidance in the form of human experience. Recommended settings for the genetic algorithm were outlined and used to great effect allowing the genetic algorithm to learn a scoring matrix which saw an increase in the

number of wins by 12% for the player using it.

As a result of a combination of the work in this thesis, our intuition about good strategies for the game of draughts was also confirmed. This includes: that going first doesn't provide either player much of an advantage; that all of the squares on the board are not to be considered equally valuable when deciding where to place pieces on the board; and furthermore that squares furthest away from the where the player starts and squares on the edge of the board seem to be more valuable squares to occupy with pieces.

8 Conclusion

8.1 Further Work / Open Questions

The work in this thesis could be continued in a variety of different and interesting ways. These include augmenting the Alpha Beta Pruning using Aspiration Search, Iterative Deepening and Killer Heuristic [21]. With Aspiration Search a narrower alpha beta search window is implemented which allows deeper searches at the cost of being less accurate. Iterative Deepening is a technique that runs numerous times, incriminating the depth with each iteration. This provides opportunity for the moves to be reordered after each iteration which can result in branches being pruned at earlier depths depending on the success of the method used to reorder the moves [21]. Killer heuristic sees the order of the moves changed in order to try and maximize the amount of pruning done. This can involve examining moves that will result in an opponent's piece being taken as a priority. Good move ordering techniques have been used successfully by algorithms such as the NegaScout. NegaScout is an algorithm which unlike the alpha beta pruning algorithm is limited to being applied only to zero-sum games, but which can be faster than alpha-beta pruning as it has been proved to never examine a node that can be pruned by alpha-beta [19]. This algorithm could also be used as another interesting source for comparison with algorithms augmented in the ways detailed in this section.

Other ways to continue the research would be to no longer insist on returning the optimal move as selected by MiniMax which would allow heuristic "early" pruning of branches that are unpromising, and so permit a greater search depth on the most promis-

ing branches. Another idea would be to wait until the game gets close to being finished and then reuse the unmodified alpha-beta pruning algorithm so the full game tree is iterated over [5]. This would provide the optimal move with which to proceed in scenarios in which the algorithm won't take as long to execute owing to the reduced number of pieces on the board.

In a similar vein alternative artificial intelligence techniques to genetic algorithms could be used to tackle the problem of learning a matrix for use in the heuristic evaluation function, such as neural networks, which could be compared with the genetic algorithm developed in this thesis. Other areas of investigation include analysing different games that are computationally intensive to solve such as Go. Attempts at matching the grand masters of the game have until now remained unsuccessful. Success on the computers part has only been achieved on boards of reduced size, or where sizable handicaps in the number of pieces have been imposed on the human opponent. This is owing to the fact that there are regularly 5 times as many possible moves as there are in chess. The most successful Go computer programs of recent times run as part of the GO game engine "Zen" developed by Yoji Ojima. Its various forms have held ranking between 3rd and 5th Dan where Dan is achieved by advanced amateur human players [15].

8.2 Bibliography

Bibliography

[1] Proceedings of the 1st international conference on genetic algorithms. Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[2] Victor Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[3] Nils Barricelli. Numerical testing of evolution theories. *Acta Biotheoretica*, 16:99–126, 1963.

[4] R. W. Beeler, M. Gosper and R. Schroeppel. Hakmem. Cambridge, MA: MIT Artificial Intelligence Laboratory, Memo AIM-239, p. 35,, FEB 1972.

[5] Kevin Anthony Cherry. An intelligent othello player combining machine learning and game specific heuristics. Master's thesis, Louisiana State University, May 2011.

[6] Oracle Corporation. Netbeans ide features. http://netbeans.org/features/index.html, website, Jan 2013.

[7] Oracle Corporation. What is java? http://java.com/en/download/faq/whatis_java.xml, website, Jan 2013.

[8] M. Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

[9] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 1997.

[10] Alexis Drogoul. When ants play chess (or can strategies emerge from tactical behaviors?). In *In Proceedings of Fifth European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 93*, pages 13–27. Springer, 1995.

[11] M. Gardner. *Hexaflexagons and other mathematical diversions: The first Scientific American book of puzzles and games.* University of Chicago Press, 1988.

[12] D. Richard Hipp. About sqlite. http://www.sqlite.org/about.html, website, Jan 2013.

[13] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, MA, USA, 1992.

[14] Kevin Leyton-Brown and Yoav Shoham. *Essentials of Game Theory: A Concise, Multidisciplinary Introduction*. Morgan and Claypool Publishers, 1st edition, 2008.

[15] Sensei's Library. Zen (go program). http://senseis.xmp.net/?ZenGoProgram, website, January 2013.

[16] Jim Loy. Opening weaknesses. http://www.jimloy.com/checkers/weakness.htm, website, 1997.

[17] John McCarthy. What is artificial intelligence? www-formal.stanford.edu/jmc/whatisai/whatisai.html, website, July 2002.

[18] Allen Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2(4):320 –335, oct. 1958.

[19] Alexander Reinefeld. *Spielbaum-Suchverfahren*, volume 200 of *Informatik-Fachberichte*. Springer, 1989.

[20] Stefan Reisch. Gobang ist pspace-vollstandig. *Acta Informatica*, 13:59–66, 1980. 10.1007/BF00288536.

[21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Upper Saddle River : Prentice Hall, third edition, Dec 2010.

[22] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210 –229, july 1959.

[23] A. L. Samuel. Some studies in machine learning using the game of checkers. ii x2014;recent progress. *IBM Journal of Research and Development*, 11(6):601 –617, nov. 1967.

[24] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[25] Steve Schaeffer. Mathematical recreations. http://www.mathrec.org/old/2002jan/solutions.html, website, 2002.

[26] Hans-Paul Schwefel. *Numerical Optimization of Computer Models*. John Wiley & Sons, Inc., New York, NY, USA, 1981.

[27] Claude E. Shannon. Xxii. programming a computer for playing chess. *Philosophical Magazine Series 7*, 41(314):256–275, 1950.

[28] Shiven Sharma, Ziad Kobti, and Scott Goodwin. General game playing with ants. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning*, SEAL '08, pages 381–390, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[30] Thomas Stutzle and Holger H. Hoos. Max-min ant system. *Future Gener. Comput. Syst.*, 16(9):889–914, June 2000.

[31] J. v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928. 10.1007/BF01448847.

[32] Vicki Wenderlich. Free game art: Checkers. http://vickiwenderlich.s3-website-us-east-1.amazonaws.com/wp-content/uploads/2012/08/mockup1_3001.jpg, website, August 2012.